AI Assisted Coding

Lab Assignment 12.5

Name : K. Charan Yadav

Hall Ticket no : 2303A52367

Batch No : 20

Task -1:

Prompt: Implement a sorting algorithm to arrange elements in ascending order efficiently regardless of input size or order. The solution should divide the dataset recursively and merge sorted sublists to produce the final sorted output.

```python
Assignment-12.5.py > merge
# Task-1:
def merge_sort(arr):
    """Sorts a list in ascending order using the Merge Sort algorithm.
    Time Complexity: O(n log n) - all cases (best, average, worst)
    Space Complexity: O(n) - requires additional space for merging
    Args:
        arr: List of comparable elements
    Returns:
        Sorted list in ascending order"""
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
def merge(left, right):
    """Merges two sorted lists into a single sorted list."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Test cases
if __name__ == "__main__":
    test_cases = [[64, 34, 25, 12, 22, 11, 90],
        [5, 2, 8, 1, 9],
        [1],
        [],
        [3, 3, 1, 2],]
    for test in test_cases:
        print(f"Original: {test}")
        print(f"Sorted:   {merge_sort(test)}\n")
```

OUTPUT:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
Original: [64, 34, 25, 12, 22, 11, 90]
Sorted:   [11, 12, 22, 25, 34, 64, 90]

Original: [5, 2, 8, 1, 9]
Sorted:   [1, 2, 5, 8, 9]

Original: [1]
Sorted:   [1]

Original: []
Sorted:   []

Original: [3, 3, 1, 2]
Sorted:   [1, 2, 3, 3]
```

Justification:

Merge Sort guarantees **O(n log n)** time complexity in all cases, making it reliable for large datasets. It follows the divide-and-conquer approach, ensuring predictable performance. The algorithm is stable and suitable when consistent execution time is required.

Task 2:

Prompt: Develop an efficient searching algorithm to locate a target element within a sorted dataset using minimum comparisons. The method should repeatedly reduce the search space.

Assignment-12.5.py > merge

```python
# Task-2:
def binary_search(arr, target):
    """Searches for a target element in a sorted list using Binary Search.
    Time Complexity: O(log n) - all cases (best, average, worst)
    Space Complexity: O(1) - only uses constant extra space
    Args:
        arr: Sorted list of comparable elements
        target: Element to search for
    Returns:
        Index of target if found, -1 otherwise"""
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Test cases
if __name__ == "__main__":
    sorted_list = [1, 5, 7, 10, 15, 20, 25, 30, 35, 40]
    test_targets = [5, 25, 1, 40, 50, 7]
    for target in test_targets:
        result = binary_search(sorted_list, target)
        if result != -1:
            print(f"Target {target}: Found at index {result}")
        else:
            print(f"Target {target}: Not found")
```

Output:

```
Target 5: Found at index 1
Target 25: Found at index 6
Target 1: Found at index 0
Target 40: Found at index 9
Target 50: Not found
Target 7: Found at index 2
Search by ID 102:
Apt(102, Bob, Dr. Johnson, 2024-01-15 10:30, $200)
```

Justification:

Binary Search reduces search space by half each iteration, achieving **O(log n)** time complexity. It is significantly faster than linear search for large sorted datasets. Constant space usage makes it memory efficient.

Task 3:

Prompt:

Design a system to manage healthcare appointments allowing search by appointment ID and sorting based on appointment time and consultation fee for administrative analysis.

```
Assignment-12.5.py > merge
class Appointment:
    def __init__(self, apt_id, patient_name, doctor_name, time, fee):
        self.fee = fee
    def __repr__(self):
        return f"Apt({self.apt_id}, {self.patient_name}, {self.doctor_name}, {self.time}, ${self.fee})"
def linear_search_by_id(appointments, target_id):
    """Searches for appointment by ID using Linear Search.
    Justification: Dataset typically small (< 1000 appointments),
    simpler implementation, acceptable O(n) performance.
    Time Complexity: O(n)
    Space Complexity: O(1)"""
    for apt in appointments:
        if apt.apt_id == target_id:
            return apt
    return None
def sort_appointments_by_time(appointments):
    """Sorts appointments by time using Quick Sort (via sorted).
    Justification: Fast O(n log n) average case, good for medium datasets,
    handles datetime comparison efficiently.
    Time Complexity: O(n log n) average
    Space Complexity: O(n)"""
    return sorted(appointments, key=lambda x: x.time)
def sort_appointments_by_fee(appointments):
    """Sorts appointments by consultation fee using Merge Sort approach.
    Justification: Stable sorting guaranteed, consistent O(n log n),
    preserves order of equal fees.
    Time Complexity: O(n log n) all cases
    Space Complexity: O(n)"""
    return sorted(appointments, key=lambda x: x.fee)
# Test cases
if __name__ == "__main__":
    appointments = [Appointment(101, "Alice", "Dr. Smith", "2024-01-15 09:00", 150),
        Appointment(102, "Bob", "Dr. Johnson", "2024-01-15 10:30", 200),
        Appointment(103, "Charlie", "Dr. Smith", "2024-01-14 14:00", 150),
        Appointment(104, "Diana", "Dr. Brown", "2024-01-15 11:00", 175),]
    print("Search by ID 102:")
    print(linear_search_by_id(appointments, 102), "\n")
    print("Sorted by Time:")
    for apt in sort_appointments_by_time(appointments):
        print(apt)
    print("\nSorted by Fee:")
    for apt in sort_appointments_by_fee(appointments):
        print(apt)
```

Output:

```
Sorted by Time:
Apt(103, Charlie, Dr. Smith, 2024-01-14 14:00, $150)
Apt(101, Alice, Dr. Smith, 2024-01-15 09:00, $150)
Apt(102, Bob, Dr. Johnson, 2024-01-15 10:30, $200)
Apt(104, Diana, Dr. Brown, 2024-01-15 11:00, $175)

Sorted by Fee:
Apt(101, Alice, Dr. Smith, 2024-01-15 09:00, $150)
Apt(103, Charlie, Dr. Smith, 2024-01-14 14:00, $150)
Apt(104, Diana, Dr. Brown, 2024-01-15 11:00, $175)
Apt(102, Bob, Dr. Johnson, 2024-01-15 10:30, $200)
Search by Ticket ID 1003:
Ticket(1003, Charlie, Train T101, Seat 8, 2024-02-15)
```

Justification:

Linear search is suitable since appointment datasets are typically small and unsorted. Sorting by time improves scheduling efficiency, while stable sorting by fee preserves equal-cost ordering. Using efficient sorting ensures quick reporting and management

Task -4:

Prompt: Create a railway ticket management module that enables searching tickets by ID and organizing records based on travel date and seat number.

```
Assignment-12.5.py > linear_search_by_ticket_id
class Ticket:
    def __init__(self, ticket_id, passenger_name, train_number, seat_number, travel_date):
        self.train_number = train_number
        self.seat_number = seat_number
        self.travel_date = travel_date
    def __repr__(self):
        return f"Ticket({self.ticket_id}, {self.passenger_name}, Train {self.train_number}, Seat {self.seat_number}, {self.travel_date})"
def linear_search_by_ticket_id(tickets, target_id):
    """Searches for ticket by ID using Linear Search.
    Justification: Small dataset (typically < 10,000 tickets in active system),
    unsorted data, simple implementation, acceptable O(n) performance.
    Time Complexity: O(n)
    Space Complexity: O(1)"""
    for ticket in tickets:
        if ticket.ticket_id == target_id:
            return ticket
    return None
def sort_tickets_by_travel_date(tickets):
    """Sorts tickets by travel date using Merge Sort approach.
    Justification: Stable sorting preserves order, consistent O(n log n),
    important for maintaining passenger list integrity.
    Time Complexity: O(n log n) all cases
    Space Complexity: O(n)"""
    return sorted(tickets, key=lambda x: x.travel_date)
def sort_tickets_by_seat_number(tickets):
    """Sorts tickets by seat number using Quick Sort (via sorted).
    Justification: Fast O(n log n) average case, efficient for numeric sorting,
    good for seat arrangement reports.
    Time Complexity: O(n log n) average
    Space Complexity: O(n)"""
    return sorted(tickets, key=lambda x: x.seat_number)
# Test cases
if __name__ == "__main__":
    tickets = [Ticket(1001, "Alice", "T101", 12, "2024-02-15"),
        Ticket(1002, "Bob", "T102", 5, "2024-02-20"),
        Ticket(1003, "Charlie", "T101", 8, "2024-02-15"),
        Ticket(1004, "Diana", "T103", 15, "2024-02-18"),]
    print("Search by Ticket ID 1003:")
    print(linear_search_by_ticket_id(tickets, 1003), "\n")
    print("Sorted by Travel Date:")
    for ticket in sort_tickets_by_travel_date(tickets):
        print(ticket)
    print("\nSorted by Seat Number:")
    for ticket in sort_tickets_by_seat_number(tickets):
        print(ticket)
```

Output:

```
Sorted by Travel Date:
Ticket(1001, Alice, Train T101, Seat 12, 2024-02-15)
Ticket(1003, Charlie, Train T101, Seat 8, 2024-02-15)
Ticket(1004, Diana, Train T103, Seat 15, 2024-02-18)
Ticket(1002, Bob, Train T102, Seat 5, 2024-02-20)

Sorted by Seat Number:
Ticket(1002, Bob, Train T102, Seat 5, 2024-02-20)
Ticket(1003, Charlie, Train T101, Seat 8, 2024-02-15)
Ticket(1001, Alice, Train T101, Seat 12, 2024-02-15)
Ticket(1004, Diana, Train T103, Seat 15, 2024-02-18)
Search by Student ID 203:
Allocation(Student 203, Room 102, Floor 1, 2024-01-08)
```

Justification:

Linear search simplifies lookup in moderately sized unsorted ticket data. Sorting by travel date helps schedule tracking, while seat sorting supports passenger arrangement reports. Efficient sorting ensures scalability and clarity in booking systems.

Task – 5:

Prompt: Develop a hostel allocation system to search student room assignments and sort allocations by room number and allocation date for administrative monitoring.

```
Assignment-12.5.py > linear_search_by_ticket_id
class RoomAllocation:
    def __init__(self, student_id, room_number, floor, allocation_date):
        self.floor = floor
        self.allocation_date = allocation_date
    def __repr__(self):
        return f"Allocation(Student {self.student_id}, Room {self.room_number}, Floor {self.floor}, {self.allocation_date})"
def linear_search_by_student_id(allocations, target_id):
    """Searches for allocation by student ID using Linear Search.
    Justification: Small dataset (typically < 5,000 students per hostel),
    unsorted data, simple implementation, acceptable O(n) performance.
    Time Complexity: O(n)
    Space Complexity: O(1)"""
    for allocation in allocations:
        if allocation.student_id == target_id:
            return allocation
    return None
def sort_allocations_by_room_number(allocations):
    """Sorts allocations by room number using Quick Sort (via sorted).
    Justification: Fast O(n log n) average case, efficient for numeric sorting,
    good for facility management and maintenance reports.
    Time Complexity: O(n log n) average
    Space Complexity: O(n)"""
    return sorted(allocations, key=lambda x: x.room_number)
def sort_allocations_by_date(allocations):
    """Sorts allocations by date using Merge Sort approach.
    Justification: Stable sorting preserves order, consistent O(n log n),
    important for tracking allocation history chronologically.
    Time Complexity: O(n log n) all cases
    Space Complexity: O(n)"""
    return sorted(allocations, key=lambda x: x.allocation_date)
# Test cases
if __name__ == "__main__":
    allocations = [RoomAllocation(201, 105, 1, "2024-01-10"),
        RoomAllocation(202, 203, 2, "2024-01-12"),
        RoomAllocation(203, 102, 1, "2024-01-08"),
        RoomAllocation(204, 301, 3, "2024-01-15"),]
    print("Search by Student ID 203:")
    print(linear_search_by_student_id(allocations, 203), "\n")
    print("Sorted by Room Number:")
    for allocation in sort_allocations_by_room_number(allocations):
        print(allocation)
    print("\nSorted by Allocation Date:")
    for allocation in sort_allocations_by_date(allocations):
        print(allocation)
```

Output:

```
Search by Student ID 203:
Allocation(Student 203, Room 102, Floor 1, 2024-01-08)

Sorted by Room Number:
Allocation(Student 203, Room 102, Floor 1, 2024-01-08)
Allocation(Student 201, Room 105, Floor 1, 2024-01-10)
Allocation(Student 202, Room 203, Floor 2, 2024-01-12)
Allocation(Student 204, Room 301, Floor 3, 2024-01-15)
```

Justification:

Linear search is practical due to limited hostel capacity datasets. Sorting by room number aids maintenance management, while date sorting maintains chronological allocation history. Stable sorting ensures reliable record tracking.

Task - 6:

Prompt: Implement a movie catalog system that allows searching movies by ID and sorting movies based on ratings and release years for recommendation and browsing features.

```python
Assignment-12.5.py > linear_search_by_ticket_id
class Movie:
    def __init__(self, movie_id, title, genre, rating, release_year):
        self.rating = rating
        self.release_year = release_year
    def __repr__(self):
        return f"Movie({self.movie_id}, {self.title}, {self.genre}, {self.rating}★, {self.release_year})"
def linear_search_by_movie_id(movies, target_id):
    """Searches for movie by ID using Linear Search.
    Justification: Movie catalog typically unsorted by ID, dataset moderate (< 50,000),
    simple implementation, acceptable O(n) performance for real-time queries.
    Time Complexity: O(n)
    Space Complexity: O(1)"""
    for movie in movies:
        if movie.movie_id == target_id:
            return movie
    return None
def sort_movies_by_rating(movies):
    """Sorts movies by rating using Quick Sort (via sorted).
    Justification: Fast O(n log n) average case, efficient for numeric sorting,
    ideal for recommendation lists and top-rated displays.
    Time Complexity: O(n log n) average
    Space Complexity: O(n)"""
    return sorted(movies, key=lambda x: x.rating, reverse=True)
def sort_movies_by_release_year(movies):
    """Sorts movies by release year using Merge Sort approach.
    Justification: Stable sorting preserves original order for ties, consistent O(n log n),
    important for chronological organization and year-based filtering.
    Time Complexity: O(n log n) all cases
    Space Complexity: O(n)"""
    return sorted(movies, key=lambda x: x.release_year)
# Test cases
if __name__ == "__main__":
    movies = [Movie(1001, "Inception", "Sci-Fi", 8.8, 2010),
        Movie(1002, "The Shawshank Redemption", "Drama", 9.3, 1994),
        Movie(1003, "The Dark Knight", "Action", 9.0, 2008),
        Movie(1004, "Pulp Fiction", "Crime", 8.9, 1994),]
    print("Search by Movie ID 1003:")
    print(linear_search_by_movie_id(movies, 1003), "\n")
    print("Sorted by Rating (Highest First):")
    for movie in sort_movies_by_rating(movies):
        print(movie)
    print("\nSorted by Release Year:")
    for movie in sort_movies_by_release_year(movies):
        print(movie)
```

Output:

```
Search by Movie ID 1003:
Movie(1003, The Dark Knight, Action, 9.0★, 2008)

Sorted by Rating (Highest First):
Movie(1002, The Shawshank Redemption, Drama, 9.3★, 1994)
Movie(1003, The Dark Knight, Action, 9.0★, 2008)
Movie(1004, Pulp Fiction, Crime, 8.9★, 1994)
Movie(1001, Inception, Sci-Fi, 8.8★, 2010)

Sorted by Release Year:
Movie(1002, The Shawshank Redemption, Drama, 9.3★, 1994)
Movie(1004, Pulp Fiction, Crime, 8.9★, 1994)
Movie(1003, The Dark Knight, Action, 9.0★, 2008)
Movie(1001, Inception, Sci-Fi, 8.8★, 2010)
```

Justification:

Movie datasets are often unsorted, making linear search simple and flexible. Sorting by rating helps generate recommendation lists, while release year sorting enables chronological filtering. Efficient sorting improves user experience.

Task - 7:

Prompt: Build a crop monitoring system capable of searching crops by ID and sorting crop data based on soil moisture and yield estimates for agricultural decision support.

```
Assignment-12.5.py > linear_search_by_ticket_id
class Crop:
    def __init__(self, crop_id, crop_name, soil_moisture, temperature, yield_estimate):
        self.soil_moisture = soil_moisture
        self.temperature = temperature
        self.yield_estimate = yield_estimate
    def __repr__(self):
        return f"Crop({self.crop_id}, {self.crop_name}, Moisture: {self.soil_moisture}%, Temp: {self.temperature}°C, Yield: {self.yield_estimate} tons)"
def linear_search_by_crop_id(crops, target_id):
    """Searches for crop by ID using Linear Search.
    Justification: Crop data typically unsorted by ID, dataset moderate (< 10,000 crops),
    simple implementation, acceptable O(n) performance for real-time monitoring queries.
    Time Complexity: O(n)
    Space Complexity: O(1)"""
    for crop in crops:
        if crop.crop_id == target_id:
            return crop
    return None
def sort_crops_by_moisture_level(crops):
    """Sorts crops by soil moisture level using Quick Sort (via sorted).
    Justification: Fast O(n log n) average case, efficient for numeric sorting,
    ideal for identifying crops needing irrigation.
    Time Complexity: O(n log n) average
    Space Complexity: O(n)"""
    return sorted(crops, key=lambda x: x.soil_moisture)
def sort_crops_by_yield_estimate(crops):
    """Sorts crops by yield estimate using Merge Sort approach.
    Justification: Stable sorting preserves order, consistent O(n log n),
    important for crop performance analysis and yield forecasting.
    Time Complexity: O(n log n) all cases
    Space Complexity: O(n)"""
    return sorted(crops, key=lambda x: x.yield_estimate, reverse=True)
# Test cases
if __name__ == "__main__":
    crops = [Crop(501, "Wheat", 45, 22, 5.2),
        Crop(502, "Rice", 75, 28, 6.8),
        Crop(503, "Corn", 50, 25, 7.1),
        Crop(504, "Barley", 40, 20, 4.5),]
    print("Search by Crop ID 503:")
    print(linear_search_by_crop_id(crops, 503), "\n")
    print("Sorted by Soil Moisture Level (Low to High):")
    for crop in sort_crops_by_moisture_level(crops):
        print(crop)
    print("\nSorted by Yield Estimate (High to Low):")
    for crop in sort_crops_by_yield_estimate(crops):
        print(crop)
```

Output:

```
Search by Crop ID 503:
Crop(503, Corn, Moisture: 50%, Temp: 25°C, Yield: 7.1 tons)

Sorted by Soil Moisture Level (Low to High):
Crop(504, Barley, Moisture: 40%, Temp: 20°C, Yield: 4.5 tons)
Crop(501, Wheat, Moisture: 45%, Temp: 22°C, Yield: 5.2 tons)
Crop(503, Corn, Moisture: 50%, Temp: 25°C, Yield: 7.1 tons)
Crop(502, Rice, Moisture: 75%, Temp: 28°C, Yield: 6.8 tons)

Sorted by Yield Estimate (High to Low):
Crop(503, Corn, Moisture: 50%, Temp: 25°C, Yield: 7.1 tons)
Crop(502, Rice, Moisture: 75%, Temp: 28°C, Yield: 6.8 tons)
Crop(501, Wheat, Moisture: 45%, Temp: 22°C, Yield: 5.2 tons)
Crop(504, Barley, Moisture: 40%, Temp: 20°C, Yield: 4.5 tons)
```

Justification:

Linear search is adequate for moderate agricultural datasets. Moisture sorting helps identify irrigation priorities, while yield sorting assists productivity analysis. Efficient sorting enables real-time monitoring insights.

Task - 8:

Prompt: Design a flight management module to search flights by ID and organize schedules based on departure and arrival times for airport operations.

```python
Assignment-12.5.py > linear_search_by_ticket_id
class Flight:
    def __init__(self, flight_id, airline_name, departure_time, arrival_time, status):
        self.arrival_time = arrival_time
        self.status = status
    def __repr__(self):
        return f"Flight({self.flight_id}, {self.airline_name}, Depart: {self.departure_time}, Arrive: {self.arrival_time}, {self.status})"
def linear_search_by_flight_id(flights, target_id):
    """Searches for flight by ID using Linear Search.
    Justification: Flight data typically unsorted by ID, dataset moderate (< 50,000 flights),
    simple implementation, acceptable O(n) performance for real-time lookup queries.
    Time Complexity: O(n)
    Space Complexity: O(1)"""
    for flight in flights:
        if flight.flight_id == target_id:
            return flight
    return None
def sort_flights_by_departure_time(flights):
    """Sorts flights by departure time using Quick Sort (via sorted).
    Justification: Fast O(n log n) average case, efficient for datetime sorting,
    ideal for departure boards and schedule displays.
    Time Complexity: O(n log n) average
    Space Complexity: O(n)"""
    return sorted(flights, key=lambda x: x.departure_time)
def sort_flights_by_arrival_time(flights):
    """Sorts flights by arrival time using Merge Sort approach.
    Justification: Stable sorting preserves order, consistent O(n log n),
    important for gate assignments and arrival sequence integrity.
    Time Complexity: O(n log n) all cases
    Space Complexity: O(n)"""
    return sorted(flights, key=lambda x: x.arrival_time)
# Test cases
if __name__ == "__main__":
    flights = [Flight("AA101", "American Airlines", "2024-02-15 08:00", "2024-02-15 11:30", "On-Time"),
        Flight("UA202", "United Airlines", "2024-02-15 09:45", "2024-02-15 13:15", "Delayed"),
        Flight("DL303", "Delta Airlines", "2024-02-15 07:30", "2024-02-15 10:45", "On-Time"),
        Flight("SW404", "Southwest Airlines", "2024-02-15 10:15", "2024-02-15 14:00", "On-Time"),]
    print("Search by Flight ID AA101:")
    print(linear_search_by_flight_id(flights, "AA101"), "\n")
    print("Sorted by Departure Time:")
    for flight in sort_flights_by_departure_time(flights):
        print(flight)
    print("\nSorted by Arrival Time:")
    for flight in sort_flights_by_arrival_time(flights):
        print(flight)
```

Output:

```
Search by Flight ID AA101:
Flight(AA101, American Airlines, Depart: 2024-02-15 08:00, Arrive: 2024-02-15 11:30, On-Time)

Sorted by Departure Time:
Flight(DL303, Delta Airlines, Depart: 2024-02-15 07:30, Arrive: 2024-02-15 10:45, On-Time)
Flight(AA101, American Airlines, Depart: 2024-02-15 08:00, Arrive: 2024-02-15 11:30, On-Time)
Flight(UA202, United Airlines, Depart: 2024-02-15 09:45, Arrive: 2024-02-15 13:15, Delayed)
Flight(SW404, Southwest Airlines, Depart: 2024-02-15 10:15, Arrive: 2024-02-15 14:00, On-Time)

Sorted by Arrival Time:
Flight(DL303, Delta Airlines, Depart: 2024-02-15 07:30, Arrive: 2024-02-15 10:45, On-Time)
Flight(AA101, American Airlines, Depart: 2024-02-15 08:00, Arrive: 2024-02-15 11:30, On-Time)
Flight(UA202, United Airlines, Depart: 2024-02-15 09:45, Arrive: 2024-02-15 13:15, Delayed)
Flight(SW404, Southwest Airlines, Depart: 2024-02-15 10:15, Arrive: 2024-02-15 14:00, On-Time)
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
```

Justification:

Flight records may not be pre-sorted, making linear search suitable for quick lookup. Sorting by departure time improves boarding schedules, while arrival sorting supports gate management. Stable sorting maintains operational accuracy.